APL contre MATLAB

Charles HUBERT

Introduction

Dans cet article je compare les deux langages de programmation APL et MATLAB. Il existe entre eux de nombreuses différences qui influent beaucoup sur leur efficacité pratique ; mais pour ne pas lasser le lecteur je me suis limité à un petit nombre de ces différences permettant de se faire une idée.

Si MATLAB n'utilise guère que les caractères ASCII, au contraire APL se sert de caractères spéciaux (souvent des idéogrammes) plus faciles à retenir que les nombreux symboles de fonctions ou les mots réservés de MATLAB.

Dans la suite, les exemples dans les deux langages seront repérés par

MATLAB et APL

Si certaines fonctions de ma composition interviennent, ces repères deviendront

MATLAB/HUB et APL/HUB

Souvent apparaîtront côte à côte une même fonction en MATLAB et en APL afin de les comparer.

Les réponses de MATLAB sont souvent bavardes ; celles d'APL ne contiennent que l'essentiel ou que les détails qu'on a demandés. La manipulation de la zone de travail (workspace) est beaucoup plus facile en APL qu'en MATLAB.

Les symboles

Les deux langages contiennent des fonctions que nous appellerons primitives parce qu'ils les connaissent sans avoir à les définir : par exemple l'addition "+".

En MATLAB les primitives qui ne sont pas représentées par un ou plusieurs caractères spéciaux le sont par des symboles construits de la même façon que les symboles créés par le programmeur (exemple "size"). Celui-ci ne peut pas utiliser ces symboles, ni les mots réservés (exemple "if"), pour autre chose.

En APL une fonction ou une variable primitive est représentée, ou bien par un caractère spécial (exemple : multiplication "×"), ou bien par un mot préfixé par le caractère spécial "⎕" (quad, exemple : date et heure "⎕ts" = time stamp). Le nom d'une fonction ou d'une variable primitive contient toujours un caractère spécial ce qui empêche tout conflit avec un objet créé par le programmeur. Il n'y a pas de mot réservé.

Pendant un travail en MATLAB, une fonction "truc" (par exemple) est un fichier du même nom "truc.m" sur le disque dur. Comme c'est le nom du fichier qui compte, la fonction pourra être appelée par "TRUC" ou "Truc" ou "tRuC",... car la gestion des fichiers confond les majuscules et les minuscules ; de plus si on a remplacé "truc" par "machin" dans la ligne "function" du fichier "truc.m", MATLAB considérera qu'il s'agit toujours de la fonction "truc" et non de la fonction "machin".

Pendant un travail en APL, une fonction "truc" (par exemple) se trouve en mémoire vive (RAM). Alors "truc" ou "TRUC" ou "Truc" ou "tRuC" sont autant de fonctions différentes car APL distingue les majuscules et les minuscules. On peut en profiter pour placer une majuscule au premier caractère de chacun des mots qui composent un symbole, ce qui améliore la lisibilité ; par exemple "RacPol" signifie Racines de Polynômes.

En MATLAB, les variables se placent à deux niveaux : local (privé) et global (public). Une variable globale est accessible partout où elle figure dans une déclaration "global". Si elle ne figure pas dans une déclaration "global", elle n'est accessible que dans la fonction où elle est affectée. La déclaration "global" équivaut à la déclaration "common" du fortran.

En APL, les variables et fonctions se placent dans une arborescence dynamique. On déclare les symboles (variables et fonctions) locaux à une fonction ("Fonc" par exemple) dans sa ligne de définition.

Les objets désignés par ces symboles sont accessibles dans "Fonc" et dans toute autre fonction que "Fonc" appelle directement ou indirectement, qui les traite alors comme globaux. Ces objets sont inaccessibles avant d'entrer dans "Fonc" ou après en être sorti. Cette organisation arborescente des symboles rend de grands services en APL. En MATLAB il faut s'en passer.

APL offre la possibilité de numéroter (indicer) les composantes d'un vecteur à partir de zéro ou un. La variable primitive "⎕io" (index origin) désigne cette origine ; on peut la choisir par "⎕io←0" ou "⎕io←1". Dans la pratique, l'origine zéro simplifie la programmation plus souvent que l'origine un.

Un algorithme simple

Le but de cet exemple est de comparer les deux versions d'une fonction qui utilise la même méthode pour calculer des nombres bien connus. Partons de la définition suivante : un nombre premier est un entier naturel dont le nombre de diviseurs est égal à 2. L'entier n>0 étant donné, il faut calculer la liste des nombres premiers ≤n en programmant cette définition. Voici les deux fonctions :

```
MATLAB :
function p=prem(n)
d=zeros(1,n) ;
for p=1:n
    i=p:p:n ; d(i)=d(i)+1 ;
end
p=find(d==2) ;
```

```
APL :
 p←Prem n;⎕io
 ⎕io←1 ⋄ p←(2=⊃+/nρ¨(-ιn)↑¨1)/ιn
```

Pour être raisonnable limitons-nous à 500 :

```
MATLAB :
>> prem(500)
ans =
  Columns 1 through 13
    2    3    5    7   11   13   17   19   23   29   31   37   41
  Columns 14 through 26
   43   47   53   59   61   67   71   73   79   83   89   97  101
  Columns 27 through 39
  103  107  109  113  127  131  137  139  149  151  157  163  167
  Columns 40 through 52
  173  179  181  191  193  197  199  211  223  227  229  233  239
  Columns 53 through 65
  241  251  257  263  269  271  277  281  283  293  307  311  313
  Columns 66 through 78
  317  331  337  347  349  353  359  367  373  379  383  389  397
  Columns 79 through 91
  401  409  419  421  431  433  439  443  449  457  461  463  467
  Columns 92 through 95
  479  487  491  499
```

```
APL :
      Prem 500
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
      101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179
      181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269
      271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367
      373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461
      463 467 479 487 491 499
```

La fonction "Prem" est plus longue en MATLAB qu'en APL ; MATLAB encombre sa réponse de commentaires inutiles.

La liste des valeurs différentes dans un tableau

Voici une fonction qui donne la liste des valeurs différentes contenues dans un tableau dans l'ordre où elles s'y trouvent.

```
MATLAB :
function y=valdif(x)
x=reshape(x,1,numel(x)) ;
y=[] ;
while length(x)~=0
    d=x(1) ; y=[y d] ;
    x=x(find(x~=d)) ;
end
```

```
APL :
 x←ValDif x
 x←((⍳⍴x)=x⍳x)/x←,x
```

Exemple :

```
MATLAB :
>> x
x =
     9    17    13     2    17
     2    10    20     4    17
     5    13    10    12    13
     5    17    13    13     5
```

```
APL :
      x
 9 17 13  2 17
 2 10 20  4 17
 5 13 10 12 13
 5 17 13 13  5
```

Les données étant rangées dans un ordre différent dans les deux langages, il faut transposer l'argument d'un côté ou de l'autre pour retrouver les résultats dans le même ordre :

```
MATLAB :
>> valdif(x)
ans =
     9     2     5    17    10    13    20     4    12
>> valdif(x')
ans =
     9    17    13     2    10    20     4     5    12
```

```
APL :
      ValDif⍉x
9 2 5 17 10 13 20 4 12
      ValDif x
9 17 13 2 10 20 4 5 12
```

La transposition s'écrit "⍉" en APL : elle fait tourner "○" la matrice autour de sa diagonale "\".
La fonction "ValDif" est plus longue en MATLAB qu'en APL.

Huit dames sur un échiquier

Il s'agit de disposer sur un échiquier huit dames mutuellement imprenables. Rappelons que l'échiquier est une table carrée de 8×8 cases ; il faut y placer huit dames (pièces du jeu) de manière qu'il y ait une dame sur chaque ligne, une dame sur chaque colonne, au plus une dame sur chaque parallèle aux deux diagonales ("\" et "/").

La fonction "dame" en MATLAB donne toutes les solutions sous la forme d'une matrice dont chaque colonne décrit une solution en indiquant, pour chaque ligne de l'échiquier, le numéro de la case que la dame doit y occuper. La fonction "Dame" en APL donne la même chose sous la forme transposée de la précédente, les cases étant numérotées à partir de zéro au lieu de un. Le côté de l'échiquier n'a pas d'importance pour l'algorithme ; on peut choisir une autre valeur que 8, c'est l'argument "n". Voici les deux fonctions :

```
MATLAB :
function d=dame(n)
d=zeros(0,1) ;
for i=1:n
    c=zeros(i,0) ;
    for j=1:size(d,2)
        x=1:n ;
        for k=1:size(d,1)
            a=d(k,j) ;
            x=x(find(a-x)) ;
            x=x(find(a+i-k-x)) ;
            x=x(find(a+k-i-x)) ;
        end
        if length(x)~=0
            c=[c [d(:,zeros(1,size(x,2))+j);x]] ;
        end
    end
    d=c ;
end
```

```
APL :
 d←Dame n;⎕io;k
 ⎕io←0 ⋄ d←,⊂⍬ ⋄ →B
A:d←⊃,/(⊂¨d),¨¨(⊂⍳n)~¨∊¨d+⊂(k-⍳k)×⊂1-⍳3
B:→(n>k←⍴↑d)⍴A ⋄ d←⊃d
```

Pour faciliter la comparaison des deux résultats, on ajoute 1 à la solution APL et on la transpose. Exemple pour la dimension 5 :

```
MATLAB :
>> dame(5)
ans =
  1  1  2  2  3  3  4  4  5  5
  3  4  4  5  1  5  1  2  2  3
  5  2  1  3  4  2  3  5  4  1
  2  5  3  1  2  4  5  3  1  4
  4  3  5  4  5  1  2  1  3  2
```

```
APL :

      ⍉1+Dame 5
 1 1 2 2 3 3 4 4 5 5
 3 4 4 5 1 5 1 2 2 3
 5 2 1 3 4 2 3 5 4 1
 2 5 3 1 2 4 5 3 1 4
 4 3 5 4 5 1 2 1 3 2
```

Une fonction "Visu" permet de visualiser ces solutions :

```
APL/HUB :
      Visu Dame 5
□ ∘ ∘ ∘ ∘│□ ∘ ∘ ∘ ∘│∘ □ ∘ ∘ ∘│∘ □ ∘ ∘ ∘│∘ ∘ □ ∘ ∘│∘ ∘ □ ∘ ∘│∘ ∘ ∘ □ ∘
∘ ∘ □ ∘ ∘│∘ ∘ ∘ □ ∘│∘ ∘ ∘ □ ∘│∘ ∘ ∘ ∘ □│□ ∘ ∘ ∘ ∘│∘ ∘ ∘ ∘ □│□ ∘ ∘ ∘ ∘
∘ ∘ ∘ ∘ □│∘ □ ∘ ∘ ∘│□ ∘ ∘ ∘ ∘│∘ ∘ □ ∘ ∘│∘ ∘ ∘ □ ∘│∘ □ ∘ ∘ ∘│∘ ∘ □ ∘ ∘
∘ □ ∘ ∘ ∘│∘ ∘ ∘ ∘ □│∘ ∘ □ ∘ ∘│□ ∘ ∘ ∘ ∘│∘ □ ∘ ∘ ∘│∘ ∘ ∘ □ ∘│∘ ∘ ∘ ∘ □
∘ ∘ ∘ □ ∘│∘ ∘ □ ∘ ∘│∘ ∘ ∘ ∘ □│∘ ∘ ∘ □ ∘│∘ ∘ ∘ ∘ □│□ ∘ ∘ ∘ ∘│∘ □ ∘ ∘ ∘

∘ ∘ ∘ □ ∘│∘ ∘ ∘ ∘ □│∘ ∘ ∘ ∘ □
∘ □ ∘ ∘ ∘│∘ □ ∘ ∘ ∘│∘ ∘ □ ∘ ∘
∘ ∘ ∘ ∘ □│∘ ∘ ∘ □ ∘│□ ∘ ∘ ∘ ∘
∘ ∘ □ ∘ ∘│□ ∘ ∘ ∘ ∘│∘ ∘ ∘ □ ∘
□ ∘ ∘ ∘ ∘│∘ ∘ □ ∘ ∘│∘ □ ∘ ∘ ∘
```

Parmi les solutions qui ne diffèrent entre elles que par une symétrie du carré, une fonction "NonSym" n'en retient qu'une :

```
APL/HUB :
      Visu NonSym Dame 5
□ ∘ ∘ ∘ ∘│∘ □ ∘ ∘ ∘
∘ ∘ □ ∘ ∘│∘ ∘ ∘ ∘ □
∘ ∘ ∘ ∘ □│∘ ∘ □ ∘ ∘
∘ □ ∘ ∘ ∘│□ ∘ ∘ ∘ ∘
∘ ∘ ∘ □ ∘│∘ ∘ ∘ □ ∘
```

La fonction "Dame" est plus longue en MATLAB qu'en APL.

La dimension des tableaux

En MATLAB les données sont des tableaux (arrays) dont la dimension comporte au moins deux nombres ; ce sont au moins des matrices. Un vecteur est une matrice ou bien à une ligne, ou bien à une colonne. Un scalaire est une matrice à une ligne et une colonne. La dimension peut comporter plus de deux nombres ; MATLAB parle alors de matrice multidimensionnelle.

En APL c'est la même chose, sauf que la dimension comporte un nombre pour un vecteur et zéro nombre pour un scalaire.

MATLAB parcourt les tableaux en faisant varier les premiers indices plus vite que les derniers (comme le fortran) : il écrit verticalement puis, quand la colonne est pleine, passe à la colonne suivante.

APL fait l'inverse : il écrit horizontalement puis, quand la ligne est pleine, passe à la ligne suivante, comme dans un texte français (ou anglais,...) normal.

Un tableau de dimension "3 4 5 6" est vu par MATLAB comme 6 paquets de 5 matrices de 4 colonnes de 3 cases, par APL comme 3 paquets de 4 matrices de 5 lignes de 6 cases. Pour visualiser le même tableau de la même façon, il faut le transformer par une des deux fonctions :

```
MATLAB :
function x=visapl(a)
r=ndims(a) ;
x=permute(a,[r-1 r r-2:-1:1]) ;
```

```
APL :
 x←VisMtl x
 x←(⌽⍋(-⍴⍴x)↑ 2 1)⍉x
```

En MATLAB "visapl" simule comment APL présente un tableau ; en APL "VisMtl" simule comment MATLAB présente un tableau. Elles sont en quelque sorte inverses l'une de l'autre. Dans les exemples "Dim" construit en APL un tableau suivant les conventions de MATLAB.

Matrice de dimension "3 5" :

```
MATLAB :
>> reshape(1:15,[3 5])
ans =
    1    4    7   10   13
    2    5    8   11   14
    3    6    9   12   15
```

```
APL/HUB :

      3 5 Dim 1+⍳99
 1 4 7 10 13
 2 5 8 11 14
 3 6 9 12 15
```

Notons qu'APL redimensionne 99 valeurs en ne prenant que les 15 premières valeurs.

Tableau de dimension "3 5 2" :

```
MATLAB :
>> reshape(1:30,[3 5 2])
ans(:,:,1) =
     1     4     7    10    13
     2     5     8    11    14
     3     6     9    12    15
ans(:,:,2) =
    16    19    22    25    28
    17    20    23    26    29
    18    21    24    27    30
```

```
APL/HUB :

      VisMtl 3 5 2 Dim 1+⍳99
  1  4  7 10 13
  2  5  8 11 14
  3  6  9 12 15

 16 19 22 25 28
 17 20 23 26 29
 18 21 24 27 30
```

```
MATLAB :          |APL : ──────────────┐
>> visapl(reshape(1:30,[3 5 2]))       ↓
ans(:,:,1) =      |     3 5 2 Dim 1+⍳99
     1    16      |   1 16
     4    19      |   4 19
     7    22      |   7 22
    10    25      |  10 25
    13    28      |  13 28
ans(:,:,2) =      |
     2    17      |   2 17
     5    20      |   5 20
     8    23      |   8 23
    11    26      |  11 26
    14    29      |  14 29
ans(:,:,3) =      |
     3    18      |   3 18
     6    21      |   6 21
     9    24      |   9 24
    12    27      |  12 27
    15    30      |  15 30
```

Tableau de dimension "3 5 3 2" :

```
MATLAB :                          |APL/HUB :
>> reshape(1:90,[3 5 3 2])        |      VisMtl 3 5 3 2 Dim 1+⍳99
ans(:,:,1,1) =                    |   1  4  7 10 13
     1     4     7    10    13    |   2  5  8 11 14
     2     5     8    11    14    |   3  6  9 12 15
     3     6     9    12    15    |
ans(:,:,2,1) =                    |  16 19 22 25 28
    16    19    22    25    28    |  17 20 23 26 29
    17    20    23    26    29    |  18 21 24 27 30
    18    21    24    27    30    |
ans(:,:,3,1) =                    |  31 34 37 40 43
    31    34    37    40    43    |  32 35 38 41 44
    32    35    38    41    44    |  33 36 39 42 45
    33    36    39    42    45    |
ans(:,:,1,2) =                    |
    46    49    52    55    58    |  46 49 52 55 58
    47    50    53    56    59    |  47 50 53 56 59
    48    51    54    57    60    |  48 51 54 57 60
ans(:,:,2,2) =                    |
    61    64    67    70    73    |  61 64 67 70 73
    62    65    68    71    74    |  62 65 68 71 74
    63    66    69    72    75    |  63 66 69 72 75
ans(:,:,3,2) =                    |
    76    79    82    85    88    |  76 79 82 85 88
    77    80    83    86    89    |  77 80 83 86 89
    78    81    84    87    90    |  78 81 84 87 90
```

Dans MATLAB on trouve les fonctions "size", "reshape" (pourquoi pas resize ?), "zeros", "ones". En s'y prenant autrement, une seule fonction "dim" aurait pu faire tout cela, et même un peu plus : c'est ce que la primitive "ρ" (rho) fait en APL. On peut programmer "dim" en MATLAB ; la voici accompagnée de la fonction "Dim" qui simule la même chose en APL en tenant compte de la convention de rangement des valeurs par MATLAB :

```
MATLAB :
function x=dim(a,d)
if nargin==1
    x=size(a) ;
else
    switch length(d)
        case 0
            d=[1 1] ;
        case 1
            d=[1 d] ;
    end
    nx=prod(d) ;
    na=numel(a) ;
    if (nx>0)&&(na==0)
        return
    end
    a=reshape(a,[1 na]) ;
    while length(a)<nx
        a=[a a] ;
    end
    x=reshape(a(1:nx),d) ;
end
```

```
APL :
 x←d Dim a
 →(⎕nc 'd')ρA
 x←((ρx)↓1 1),x←ρa ⋄ →0
A:x←⍉((⌽d),(ρd←,d)↓1 1)ρ⍉a
```

Quand on travaille en APL, on n'a pas besoin de "Dim" puisque avec son arrangement des données la primitive "ρ" suffit :

```
MATLAB :
function x=dimapl(a,d)
if nargin==1
    x=size(a) ;
else
    d=d(length(d):-1:1) ;
    r=length(d) ;
    switch r
        case 0
            d=[1 1] ;
        case 1
            d=[d 1] ;
    end
    x=permute(dim(permute(a,ndims(a):-1:1),d),length(d):-1:1) ;
end
```

```
APL :
x←dρa
```

Quand on l'appelle avec un argument, "dim" donne sa dimension, c'est ce que "size" fait. Considérons :

```
MATLAB :
>> a=2:2:24
a =
  2   4   6   8  10  12  14  16  18  20  22  24
```

```
APL :
      ⎕←a←2×1+⍳12
2 4 6 8 10 12 14 16 18 20 22 24
```

alors :

```
MATLAB :
>> size(a)
ans =
     1    12
>> dim(a)
ans =
     1    12
```

```
APL/HUB :
      Dim a
1 12
```

```
APL :
      ρa
12
```

Changeons la dimension de "a" :

```
MATLAB :
>> reshape(a,[3 4])
ans =
    2    8   14   20
    4   10   16   22
    6   12   18   24
```

mais :

```
MATLAB :
>> reshape(a,[3 5])
??? Error using ==> reshape
To RESHAPE the number of elements must not change.
```

"reshape" exige que l'argument "a" contienne le même nombre de valeurs que le résultat.

Quand on l'appelle avec deux arguments, "dim" construit une matrice dont la dimension est décrite par le deuxième argument, en prenant cycliquement ses valeurs dans le premier argument ; "dim" n'exige pas que l'argument "a" contienne le même nombre de valeurs que le résultat. Pour faire la même chose il faut dans un langage ou dans l'autre retourner la dimension et transposer l'argument et le résultat, ce que la fonction "Dim" fait :

```
MATLAB :
>> dim(a,[3 5])
ans =
    2    8   14   20    2
    4   10   16   22    4
    6   12   18   24    6
```

```
APL :
      3 5 Dim a
 2  8 14 20 2
 4 10 16 22 4
 6 12 18 24 6
```

```
APL :
      ⍉5 3ρ⍉a
 2  8 14 20 2
 4 10 16 22 4
 6 12 18 24 6
```

ou bien :

```
MATLAB :
>> dim(a,[5 3])'
ans =
    2    4    6    8   10
   12   14   16   18   20
   22   24    2    4    6
```

```
APL :
      ⍉5 3 Dim a
  2  4  6  8 10
 12 14 16 18 20
 22 24  2  4  6
```

```
APL :
      3 5 ρa
  2  4  6  8 10
 12 14 16 18 20
 22 24  2  4  6
```

ou bien :

```
MATLAB :
>> dimapl(a,[3 5])
ans =
    2    4    6    8   10
   12   14   16   18   20
   22   24    2    4    6
```

```
APL :
      3 5ρa
  2  4  6  8 10
 12 14 16 18 20
 22 24  2  4  6
```

ou encore :

```
MATLAB :
>> dim(a,[5 7])
ans =
    2   12   22    8   18    4   14
    4   14   24   10   20    6   16
    6   16    2   12   22    8   18
    8   18    4   14   24   10   20
   10   20    6   16    2   12   22
```

```
APL :
      5 7 Dim a
  2 12 22  8 18  4 14
  4 14 24 10 20  6 16
  6 16  2 12 22  8 18
  8 18  4 14 24 10 20
 10 20  6 16  2 12 22
```

Avec "dim", les fonctions "zeros" et "ones" deviennent superflues :

```
|MATLAB :          |MATLAB :          |APL :
|>> ones(3,4)      |>> dim(1,[3 4])   |
|ans =             |ans =             |      3 4⍴1
|   1   1   1   1  |   1   1   1   1  | 1 1 1 1
|   1   1   1   1  |   1   1   1   1  | 1 1 1 1
|   1   1   1   1  |   1   1   1   1  | 1 1 1 1
```

```
|MATLAB :          |MATLAB :          |APL :
|>> zeros(3,4)     |>> dim(0,[3 4])   |
|ans =             |ans =             |      3 4⍴0
|   0   0   0   0  |   0   0   0   0  | 0 0 0 0
|   0   0   0   0  |   0   0   0   0  | 0 0 0 0
|   0   0   0   0  |   0   0   0   0  | 0 0 0 0
```

```
|MATLAB :          |MATLAB :          |APL :
|>> zeros(3,4)+24  |>> dim(24,[3 4])  |
|ans =             |ans =             |      3 4⍴24
|  24  24  24  24  |  24  24  24  24  | 24 24 24 24
|  24  24  24  24  |  24  24  24  24  | 24 24 24 24
|  24  24  24  24  |  24  24  24  24  | 24 24 24 24
```

Ces exemples montrent qu'une fonction MATLAB bien conçue aurait pu en remplacer plusieurs pour faire la même chose avec moins de limitation.

Le produit des polynômes

Dans MATLAB il existe une fonction "conv" qui multiple deux polynômes. Pourquoi "conv" ? Parce que le produit de deux polynômes se ramène à un produit de convolution. Par souci de pédagogie, je préfère dire l'inverse. En APL/HUB c'est "PolMul". Mais "conv" ne multiplie que deux polynômes ; au-delà il faut itérer l'opération :

```
|MATLAB :                     |APL/HUB :
|>> a=[4 0 1] ;               |      a←4 0 1
|>> b=[-1 3] ;                |      b←¯1 3
|>> c=[5 3 5 1] ;             |      c←5 3 5 1
|>> d=[4 -2 7] ;              |      d←4 ¯2 7
|>> conv(a,b,c,d)             |      PolMul/a b c d
|??? Error using ==> conv     | ¯80 232 ¯192 586 5 500 96 92 21
|Too many input arguments.    |
```

Il faut donc écrire :

```
|MATLAB :                                          |APL/HUB :
|>> conv(a,conv(b,conv(c,d)))                      |
|ans =                                             |   a PolMul b PolMul c PolMul d
| -80  232 -192  586   5  500   96   92   21      |¯80 232 ¯192 586 5 500 96 92 21
```

ou programmer une boucle :

```
|MATLAB :                  |APL :
|function p=prodpoly(a)    |p←PolMul/a
|p=1 ;                     |
|for k=1:length(a)         |
|    p=conv(p,a{k}) ;      |
|end                       |
```

mais l'argument de "prodpoly" doit être un vecteur de cellules.

Cet exemple révèle l'impossibilité en MATLAB de répéter simplement une opération "conv" sur plusieurs données "a b c d". En APL la fonction "prodpoly" est superflue car la primitive "/" distribue la fonction "PolMul" entre les cases du vecteur gigogne "a" sans avoir rien à programmer.

Et la somme des polynômes ?

En MATLAB il faut programmer une fonction, par exemple ;

```
MATLAB :                      APL :
function s=sompoly(a)         s←+/⊃a
s=[0;0] ;
for k=1:length(a)
    b=a{k} ;
    s(k,1:length(b))=fliplr(b) ;
end
s=fliplr(sum(s)) ;
```

Le choix par MATLAB de ranger les coefficients suivant les puissances décroissantes est maladroit. Afin de placer les coefficients de la même puissance sur la même colonne la fonction "sompoly" range les coefficients de chaque polynôme suivant les puissances croissantes par "fliplr", puis, en fin de calcul, range les coefficients de la somme suivant les puissances décroissantes par "fliplr" pour que le résultat respecte la convention de MATLAB, comme les cases de l'argument. Notons aussi que "sum" fait la somme des lignes d'une matrice, sauf si elle ne contient qu'une ligne auquel cas elle fait la somme des colonnes ; c'est pour contourner cette exception gênante que "sompoly" initialise la copie des polynômes à deux lignes nulles.

En APL, où on range les coefficients suivant les puissances croissantes, on peut construire facilement l'instruction qui calcule la somme, "DISP" étant une fonction APL/HUB qui visualise la structure d'un tableau :

```
APL :                                       =>                           =>
      DISP a b c d                               ⊃a b c d           +/⊃a b c d
┌→──────────────────────────────────────┐    4  0 1 0           12 4 13 1
│┌→────┐ ┌→───┐ ┌→──────┐ ┌→──────┐    │   ¯1  3 0 0
││4 0 1│ │¯1 3│ │5 3 5 1│ │4 ¯2 7 │    │    5  3 5 1
│└~────┘ └~───┘ └~──────┘ └~──────┘    │    4 ¯2 7 0
└∊──────────────────────────────────────┘
```

Pour ajouter ou multiplier un nombre quelconque de polynômes en MATLAB il a fallu programmer deux fonctions ; en APL il n'y a rien à programmer, quelques caractères suffisent.

Le produit matriciel dans les tableaux

En MATLAB on peut calculer le produit de deux matrices par la notation "a*b". Mais cela ne s'étend pas à des tableaux de plus de deux dimensions.

En APL ce produit s'écrit "a+.×b" et fonctionne sur des tableaux de dimensions quelconques. Par exemple si "a" a 3 dimensions et "b" en a 4, on aura :

$$(a+.\times b)_{ijlmn} = \sum_k a_{ijk}\, b_{klmn}$$

Voici une fonction "tabprod" qui, en MATLAB, fait la même chose que la combinaison de primitives "+.×" d'APL :

```
MATLAB :
function x=tabprod(a,b)
da=size(a) ; ra=length(da) ;
na=[prod(da([1:ra-2])) da(ra-1) da(ra)] ;
a=reshape(a,na) ;
db=size(b) ; rb=length(db) ;
nb=[db(1) db(2) prod(db([3:rb]))] ;
b=reshape(b,nb) ;
x=zeros([na(1) na(2) nb(2) nb(3)]) ;
for ia=1:na(1)
    for ib=1:nb(3)
        x(ia,:,:,ib)=squeeze(a(ia,:,:))*b(:,:,ib) ;
    end
end
x=reshape(x,[da(1:ra-1) db(2:rb)]) ;
```

```
APL :
x←a+.×b
```

Par exemple, considérons un premier argument :

```
MATLAB :
>> a
a(:,:,1) =
    80   -87   -88
    20    79     5
a(:,:,2) =
    -3   -54    37
   -12    38   -52
a(:,:,3) =
   -98    67   -93
   -14   -91    58
a(:,:,4) =
   -84    54   -21
   -16     7    15
a(:,:,5) =
    88   -83     8
   -43   -34    52
```

```
APL :

      VisMtl a
 80 ¯87 ¯88
 20  79   5

 ¯3 ¯54  37
¯12  38 ¯52

¯98  67 ¯93
¯14 ¯91  58

¯84  54 ¯21
¯16   7  15

 88 ¯83  8
¯43 ¯34  52
```

puis un deuxième argument :

```
MATLAB :
>> b
b(:,:,1) =
    69   -36   -18   -34
   -72   -96   -14    -3
     1    59     4    55
   -60    -1   -15    54
     8   -25    40   -95
b(:,:,2) =
    12    78   -65    18
    26   -22   -65    71
   -70   -22    63     5
   -39    84    20   -68
   -87   -74    86   -68
b(:,:,3) =
    91   -69    40    20
   -33   -48   -25    43
   -43    -5    57    98
    13     9   -46   -74
    65   -29    35    -6
```

```
APL :

      VisMtl b
 69 ¯36 ¯18 ¯34
¯72 ¯96 ¯14  ¯3
  1  59   4  55
¯60  ¯1 ¯15  54
  8 ¯25  40 ¯95

 12  78 ¯65  18
 26 ¯22 ¯65  71
¯70 ¯22  63   5
¯39  84  20 ¯68
¯87 ¯74  86 ¯68

 91 ¯69  40  20
¯33 ¯48 ¯25  43
¯43  ¯5  57  98
 13   9 ¯46 ¯74
 65 ¯29  35  ¯6
```

et calculons le produit défini comme en APL :

```
MATLAB/HUB :
>>  ab=tabprod(a,b) ;
>> size(a)
ans =
    2    3    5
>> size(b)
ans =
    5    4    3
>> size(ab)
ans =
    2    3    4    3
```

```
APL :
      ab←a+.×b
      ⍴a
2 3 5
      ⍴b
5 4 3
      ⍴ab
2 3 4 3
```

Comparons les résultats :

```
MATLAB :
>> ab
ab(:,:,1,1) =
     11382  -5952  -7505
      2846   1932   3663
ab(:,:,2,1) =
    -10490  14290  -6050
       697 -11018   6919
ab(:,:,3,1) =
      2990  -1540   1329
     -1728  -3783   2725
ab(:,:,4,1) =
    -20997  17606  -4128
      1807  -4197   -954
ab(:,:,1,2) =
      3362  -2023   6539
      5273  10991 -10461
ab(:,:,2,2) =
     -5106   3606  -7988
      3970  10432  -2330
ab(:,:,3,2) =
     -5291   7328  -2276
     -5420 -16122  11481
ab(:,:,4,2) =
       465  -3093   1462
      3450   5501  -7868
ab(:,:,1,3) =
     16221 -13709  -4983
      -185   7729   3252
ab(:,:,2,3) =
     -8194  11153   4340
       369  -5771    488
ab(:,:,3,3) =
      4633  -3700  -8500
      -467  -4489   5936
ab(:,:,4,3) =
     -2445   -994  -7777
       -46  -6018   2126
```

```
APL :
      VisMtl ab
  11382  ¯5952  ¯7505
   2846   1932   3663

 ¯10490  14290  ¯6050
    697 ¯11018   6919

   2990  ¯1540   1329
  ¯1728  ¯3783   2725

 ¯20997  17606  ¯4128
   1807  ¯4197   ¯954


   3362  ¯2023   6539
   5273  10991 ¯10461

  ¯5106   3606  ¯7988
   3970  10432  ¯2330

  ¯5291   7328  ¯2276
  ¯5420 ¯16122  11481

    465  ¯3093   1462
   3450   5501  ¯7868


  16221 ¯13709  ¯4983
   ¯185   7729   3252

  ¯8194  11153   4340
    369  ¯5771    488

   4633  ¯3700  ¯8500
   ¯467  ¯4489   5936

  ¯2445   ¯994  ¯7777
    ¯46  ¯6018   2126
```

Si le produit de matrices s'écrit ''+.×'' en APL, c'est parce que chaque valeur du résultat est une somme ''+'' de produits ''×''. C'est un cas particulier d'un concept général : celui de "f.g" où "f" et "g" sont deux fonctions quelconques calculant sur deux scalaires. Par exemple la combinaison ''×.⋆'' calcule des produits ''×'' de puissances ''⋆'' ; si on veut calculer :

$$p = a\,x^2\,y + b\,x\,y^2 + c\,x\,y\,z + d\,x\,z^2 + e\,z^3$$

on peut construire la matrice des exposants :

```
APL :
      ⎕←n←⊃[0](2 1)(1 2)(1 1 1)(1 0 2)(0 0 3)
 2 1 1 1 0
 1 2 1 0 0
 0 0 1 2 3
```

placer dans les variables "x" et "a" les valeurs

```
                                  ┌           ┐
    ┌           ┐                 │ a . . .   │
    │ x  y  z   │                 │ b . . .   │
x ← │ .  .  .   │         a ←     │ c . . .   │
    │ .  .  .   │                 │ d . . .   │
    └           ┘                 │ e . . .   │
                                  └           ┘
```

et l'expression

```
(x×.*n)+.×a
```

calcule alors "p".

De nombreuses fonctions mathématiques de MATLAB sont restreintes aux seules matrices.

La structure des données

En MATLAB les données sont des tableaux (arrays) dont la dimension comporte au moins deux nombres ; ce sont au moins des matrices. Un vecteur est une matrice, ou bien à une ligne, ou bien à une colonne. Un scalaire est une matrice à une ligne et une colonne. La dimension peut comporter plus de deux nombres ; MATLAB parle alors de matrice multidimensionnelle. Toutes les cases d'un même tableau doivent être de même type : logique, nombre, caractère, structure ou cellule (tableau) et ce type ne change que si on réaffecte le tableau tout entier. Si on affecte dans un tableau des valeurs de type différent, MATLAB convertit cette valeur s'il sait le faire, c'est-à-dire entre logique, nombre et caractère. Si les cases d'un tableau contiennent des caractères on peut y faire des opérations mathématiques : ajouter 10 aux caractères ! les multiplier par 3 ! les diviser par 2 ! etc... Si les cases d'un tableau contiennent des cellules ou des structures les opérations mathématiques sont interdites au tableau entier ; pour cela il faut sortir du tableau le contenu de chaque case pour faire le calcul.

En APL les données sont aussi des tableaux, mais la dimension comporte un seul nombre pour un vecteur et zéro nombre pour un scalaire, et aussi deux nombres pour une matrice, trois nombres pour un paquet de matrices, etc... Chaque case peut contenir, indépendamment des autres cases, un nombre, ou un caractère, ou un tableau ; dans ce dernier cas il s'agit d'un tableau gigogne ; ainsi les cases d'un même tableau peuvent être de types différents. Si on affecte dans un tableau des valeurs de type différent, aucune conversion n'intervient. Il n'y a donc qu'une seule structure de donnée, qui permet de faire tout ce qui nécessite des structures différentes en MATLAB. Toutes les fonctions primitives d'APL s'appliquent à ces tableaux, à tous les niveaux, sous réserve de compatibilité de dimension, logique, mathématique (on ne peut pas ajouter 10 à des caractères...), etc...

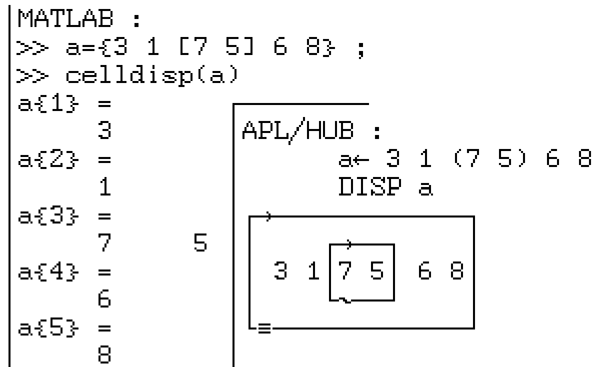
Exemple de conversion de caractères en nombres :

```
MATLAB/HUB :                          APL :
>> a=dim([2 3],12)
a =                                          ⎕←a←12⍴2 3
 2 3 2 3 2 3 2 3 2 3 2 3              2 3 2 3 2 3 2 3 2 3 2 3
>> a(3:9)='bonjour' ;                        a[2↓⍳9]←'bonjour'
>> a(3:9)                                    a[2↓⍳9]
ans =                                 bonjour
  98 111 110 106 111 117 114                a
>> a                                  2 3 bonjour 3 2 3
a =
  2   3  98 111 110 106 111 117 114   3   2   3
```

Exemple de conversion de nombres en caractères :

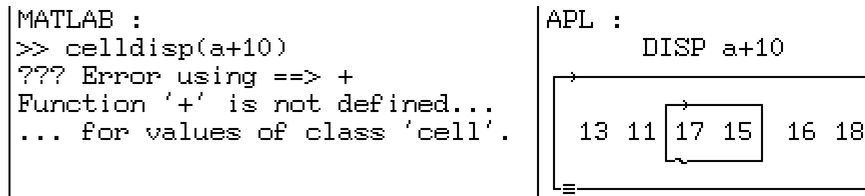
```
MATLAB/HUB :           APL :
>> a=dim('/\',12)
a =                          ⎕←a←12⍴'/\'
/\/\/\/\/\/\           /\/\/\/\/\/\
>> a(3:9)=73:79 ;            a[2↓⍳9]←73↓⍳80
>> a(3:9)
ans =                        a[2↓⍳9]
IJKLMNO                73 74 75 76 77 78 79
>> a
a =                          a
/\IJKLMNO\/\           /\ 73 74 75 76 77 78 79 \/\
```

Exemples d'interdiction ; soit une variable :


```
MATLAB :
>> a={3 1 [7 5] 6 8} ;
>> celldisp(a)
a{1} =
     3
a{2} =
     1
a{3} =
     7     5
a{4} =
     6
a{5} =
     8

APL/HUB :
       a← 3 1 (7 5) 6 8
       DISP a
 3 1 |7 5| 6 8
```

elle ne contient que des nombres auxquels MATLAB ne peut pas ajouter 10 :


```
MATLAB :
>> celldisp(a+10)
??? Error using ==> +
Function '+' is not defined...
... for values of class 'cell'.

APL :
      DISP a+10
 13 11 |17 15| 16 18
```

Soit une autre variable :

```
MATLAB :
>> b={20 10 20 70 [50 60 30]} ;
>> celldisp(b)
b{1} =
    20
b{2} =
    10
b{3} =
    20
b{4} =
    70
b{5} =
    50    60    30
```

```
APL :
b← 20 10 20 70 (50 60 30)
      DISP b
┌→─────────────────────────┐
│ 20 10 20 70 ┌→───────┐   │
│             │50 60 30│   │
│             └~───────┘   │
└∊─────────────────────────┘
```

Ces deux variables ne contiennent que des nombres, mais MATLAB ne peut pas les ajouter :

```
MATLAB :
>> celldisp(a+b)
??? Error using ==> +
Function '+' is not defined...
... for values of class 'cell'.
```

```
APL :
      DISP a+b
┌→──────────────────────────────────┐
│ 23 11 ┌→────┐ 76 ┌→───────┐       │
│       │27 25│    │58 68 38│       │
│       └~────┘    └~───────┘       │
└∊──────────────────────────────────┘
```

tandis qu'APL sait faire tout cela.

L'utilisation convenable de cette propriété des données APL dans la programmation de la fonction "PolMul" permet la répercussion d'une valeur multiple sur le produit des polynômes :

```
APL :
      DISP a←2 4 3 (4 5 6) 2 1
┌→───────────────────┐
│ 2 4 3 ┌→────┐ 2 1  │
│       │4 5 6│      │
│       └~────┘      │
└∊───────────────────┘
```

```
APL :
      DISP b←1 3 2
┌→────┐
│1 3 2│
└~────┘
```

c'est-à-dire :

$$a = 2 + 4x + 3x^2 + \underbrace{[4\ 5\ 6]}_{\text{valeurs possibles}}x^3 + 2x^4 + x^5 \qquad b = 1 + 3x + 2x^2$$

d'où le produit :

```
APL/HUB :
      DISP a PolMul b
┌→─────────────────────────────────────────────────┐
│ 2 10 19 ┌→───────┐ ┌→───────┐ ┌→───────┐ 7 2     │
│         │21 22 23│ │20 23 26│ │15 17 19│         │
│         └~───────┘ └~───────┘ └~───────┘         │
└∊─────────────────────────────────────────────────┘
```

Ce travail est plus lourd en MATLAB, puisqu'il faut créer trois polynômes "a" et les multiplier par "b" un par un dans une boucle de programme.

L'unique structure des données APL est plus puissante et plus facile à manier que les diverses structures de données MATLAB.

Un polynôme et ses racines

MATLAB décrit un polynôme suivant les puissances décroissantes ; alors :
"poly" construit un polynôme dont on donne les racines,
"roots" calcule les racines d'un polynôme.
Construisons ainsi un polynôme de degré 6 :

```
MATLAB :
>> poly([-4 -3 1 2 3 4])
ans =
     1    -3   -23    75    94 -432   288
```

puis calculons ses racines :

```
MATLAB :
>> roots(poly([-4 -3 1 2 3 4]))
ans =
   -4.0000
   -3.0000
    4.0000
    3.0000
    2.0000
    1.0000
```

APL/HUB décrit un polynôme suivant les puissances croissantes : ainsi l'indice, en origine zéro, de chaque coefficient est égal à l'exposant de la variable pour le terme correspondant. Quand des nombres complexes sont susceptibles d'intervenir on les représente par leurs deux composantes ; on représente les vecteurs réels soit par une colonne (partie réelle), soit par deux, la deuxième (partie imaginaire) étant nulle. "PolRac" calcule un Polynôme (ou plusieurs) dont on donne les Racines : "RacPol" calcule les Racines d'un Polynôme (ou plusieurs). On obtient alors :

```
APL/HUB :
      PolRac Col ¯4 ¯3 1 2 3 4
  288 0
 ¯432 0
   94 0
   75 0
  ¯23 0
   ¯3 0
    1 0
```

```
APL/HUB :
      RacPol PolRac Col ¯4 ¯3 1 2 3 4
  1 0
  2 0
  3 0
  4 0
 ¯3 0
 ¯4 0
```

Jusqu'ici tout va bien.

En MATLAB "ones" construit une matrice pleine de "1" ; faisons le même exercice pour un polynôme dont les 6 racines valent 1 :

```
MATLAB :
>> ones(1,6)
ans =
   1    1    1    1    1    1
>> poly(ones(1,6))
ans =
   1   -6   15   -20   15   -6    1
>> roots(poly(ones(1,6)))
ans =
   1.0042 + 0.0025i
   1.0042 - 0.0025i
   1.0000 + 0.0049i
   1.0000 - 0.0049i
   0.9958 + 0.0024i
   0.9958 - 0.0024i
```

tandis que :

```
|APL :       =>APL/HUB :          =>
|      6 1ρ1 |       PolRac 6 1ρ1 |       RacPol PolRac 6 1ρ1
| 1          |    1 0             | 1 0
| 1          |   ¯6 0             | 1 0
| 1          |   15 0             | 1 0
| 1          |  ¯20 0             | 1 0
| 1          |   15 0             | 1 0
| 1          |   ¯6 0             | 1 0
|            |    1 0             |
```

L'algorithme de MATLAB s'en tire bien quand c'est facile, les choses se gâtent quand c'est difficile ; celui d'APL/HUB calcule juste dans tous les cas.

Considérons maintenant le tableau à trois dimensions "p" qui contient deux polynômes et exécutons :

```
|MATLAB :                 |APL/HUB :
|>> p                     |      RacPol Col⌽⊃(1 ¯3 2)(1 ¯7 12)
|p(:,:,1) =               | 1 0
|     1    -3     2       | 2 0
|p(:,:,2) =               |
|     1    -7    12       | 3 0
|>> roots(p)              | 4 0
|ans =                    |      4⍕RacPol Col⌽ 1 ¯3 2 1 ¯7 12
|  -1.4245                |¯1.4245  0.0000
|   0.3012 + 1.4661i      | 1.9111  0.3287
|   0.3012 - 1.4661i      | 1.9111 ¯0.3287
|   1.9111 + 0.3287i      | 0.3012 ¯1.4661
|   1.9111 - 0.3287i      | 0.3012  1.4661
```

Au lieu de calculer les racines des deux polynômes :

$$1\,x^2 - 3\,x + 2 \quad \text{et} \quad 1\,x^2 - 7\,x + 12$$

il met bout à bout leurs coefficients, sans broncher, et calcule les racines du polynôme ainsi obtenu :

$$1\,x^5 - 3\,x^4 + 2\,x^3 + 1\,x^2 - 7\,x + 12$$

comme le prouve le calcul en APL/HUB. Absurde !

MATLAB ne prend pas assez de précautions pour garantir la précision de certains calculs ; il peut faire des calculs qui n'ont aucun sens.

Fonctions de MATLAB superflues en APL

Il existe dans MATLAB de nombreuses fonctions primitives dont APL se passe sans inconvénient.

D'abord APL a choisi d'abandonner les conventions d'écriture mathématique. C'est une habitude qu'on prend assez vite.

Toutes les fonctions ont la même syntaxe. Il y a trois types de fonctions : à zéro, un ou deux arguments. Par exemple si "F0" n'a pas d'argument, "F1" en a un "x" et "F2" en a deux "x" et "y", on les appelle par les expressions :

```
APL :
    F0
    F1 x
  y F2 x
```

On a la correspondance :

```
MATLAB :     APL :
 f1(a,b,c)    F1 a b c
```

et ici la fonction "F1" en APL a un argument droit subdivisé en trois sous-arguments "a", "b", "c". Dans l'expression :

```
APL :
 a b F2 c d e
```

la fonction "F2" a deux sous-arguments gauches "a", "b" et trois sous-arguments droits "c", "d", "e" ; il n'y a pas de correspondance en MATLAB où il faut se débrouiller autrement. Tout comme le signe de multiplication "×" se place entre deux nombres, la fonction "PolMul" se place entre deux polynômes :

```
MATLAB :     APL :
 a .*   b     a     ×   b
 └──┼─┐ |     |     |   |
 conv(p,q)    p PolMul  q
```

En APL la syntaxe est la même ; en MATLAB quand on passe des nombres aux polynômes, il faut changer de syntaxe.

Ensuite APL a abandonné toute priorité d'exécution, comme multiplication avant addition ; seule subsiste la priorité des parenthèses. Pour diverses raisons, une instruction ou l'intérieur d'une parenthèse s'exécutent de droite à gauche :

```
APL :        APL :            APL :          APL :
     5-3×8        (5-3)×8          3×8-5          (3×8)-5
¯19          16               9              19
```

Si MATLAB, fortran, C... exécutaient vraiment de gauche à droite, on n'écrirait pas

s = a*x + b*y + c*z

mais

a*x + b*y + c*z = s

comme avec une calculette.

Puis APL dispose de primitives, appelées opérateurs, qui prennent une ou plusieurs fonctions comme arguments. Ainsi la réduction "/" répète sa fonction de gauche entre les colonnes (suivant la dernière dimension) de son argument droit ; nous avons vu cela dans :

```
APL :
PolMul/a b c d <=> a PolMul b PolMul c PolMul d
```

La réduction "⌿" fait la même chose suivant la première dimension. Exemples de fonctions MATLAB superflues en APL :

```
MATLAB :       APL :
sum(a,1)       +⌿a
sum(a,2)       +/a
prod(a,1)      ×⌿a
prod(a,2)      ×/a
```

et :

```
MATLAB :       APL :      MATLAB :         APL :
max(a,b)       a⌈b        max(a,[],1)      ⌈⌿a
                     =>   max(a,[],2)      ⌈/a
min(a,b)       a⌊b        min(a,[],1)      ⌊⌿a
                          min(a,[],2)      ⌊/a
```

Le balayage "\" ou "⍀" permet par exemple :

```
MATLAB :         APL :
cumsum(a,1)      +⍀a
cumsum(a,2)      +\a
cumprod(a,1)     ×⍀a
cumprod(a,2)     ×\a
```

De nombreuses fonctions de MATLAB sont superflues en APL ; celui-ci les remplace par des combinaisons simples de primitives qui indiquent clairement ce qu'elles font.

Fonctions primitives APL absentes de MATLAB

Il existe dans APL des fonctions sans équivalent dans MATLAB. Il y a d'abord les fonctions qui travaillent sur les tableaux gigognes. Nous en avons vu un exemple à propos de la somme des polynômes ; la primitive "⊃" ouvre les cases d'un tableau, transformant un vecteur en matrice :

```
APL :

              ┌ a . . . ┐
⊃ a b c =>    │ b . . . │
              └ c . . . ┘
```

Voyons une autre primitive APL sans équivalent dans MATLAB.

La primitive "⍳" (iota) donne la première occurence d'une valeur dans un vecteur :

```
APL :
      a
25 20 44 13 34 20 16 13 18 27 33 22
      ⎕io←1
      x                   a⍳x
 16 13 22 34 44      7  4 12  5  3
 22 33 34 34 34     12 11  5  5  5
  7 13 20 20  7     13  4  2  2 13
 22 13 18 27  7     12  4  9 10 13
```

Ici "a" est un vecteur de dimension 12 et l'instruction "⎕io←1" sert à adopter l'origine 1 pour les indices, comme en MATLAB. 16 de x est à la 7ème position dans a, 13 est pour la première fois à la 4ème, 22 à la 12ème,... 7 à la 13ème (au delà de "a") ...

Cette primitive "⍳" fonctionne avec des vecteurs gigognes :

```
APL :
      a
 Paul Jean Pierre Jacques Michel
      x
 Michel Pierre Jacques Lucien Jean Paul
      a⍳x
5 3 4 6 2 1
```

Les fonctions de l'automatique

La "Control System Toolbox" contient des fonctions spécialisées à l'automatique. Pour se conformer à une certaine conception en vogue dans ce domaine, chaque case d'un tableau peut contenir une fraction rationnelle ; mais MATLAB l'appelle "fonction de transfert" (transfert function). On peut appeler la variable "s" ou "p" ou "z", mais pas "x" par exemple.

Transformée inverse de Laplace (automatique)

Pour calculer la transformée inverse de Laplace d'une fraction rationnelle F(p), il faut faire un détour : traiter F(p) comme la fonction de transfert d'un système linéaire invariant et lui appliquer une impulsion de Dirac. Prenons les deux fractions :

$$F_1(p) = \frac{8}{\ln 2 + p} \qquad F_2(p) = \frac{8}{\ln 2 + p + 10^{-15} p^2}$$

La valeur 10^{-15} est susceptible d'être cachée dans une description compliquée. La transformée inverse de F1(p) est exactement :

$$f_1(t) = 8\, e^{-t \ln 2} = 8 \times 2^{-t}$$

Notons $-\alpha$ et $-\beta$ les pôles de F2(p), on a avec une bonne précision :

$$\alpha = 10^{15} - \ln 2 \qquad \beta = \left[1 + 10^{-15} \ln 2\right] \ln 2$$

sa transformée inverse est :

$$f_2(t) = 8\left[1 + 2\times 10^{-15} \ln 2\right]\left[e^{-\beta t} - e^{-\alpha t}\right]$$

Pour $t \gg 10^{-15}$ le terme $e^{-\alpha t}$ est négligeable, β est très proche de ln 2 ; f2(t) est donc très proche de f1(t). Par le choix du coefficient ln 2 on voit que :

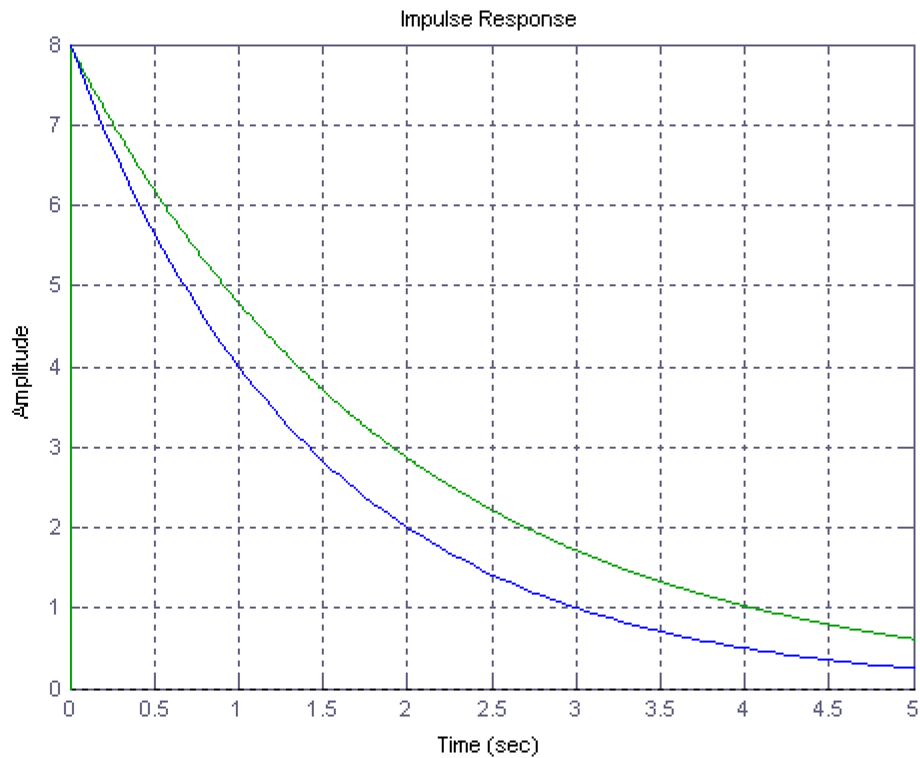
$$f_1([0\ 1\ 2\ 3\ 4\ 5]) = [8\ 4\ 2\ 1\ 0.5\ 0.25]$$

Nous ajoutons un terme du second degré nul au dénominateur de F1 pour ramener cette fraction à la

même forme que F2.

En MATLAB on fait les deux calculs séparément :

```
MATLAB :
>> hold on
>> impulse(tf(8,[0      1 log(2)]),5) ;
>> impulse(tf(8,[1E-15 1 log(2)]),5) ;
>> grid ;
```



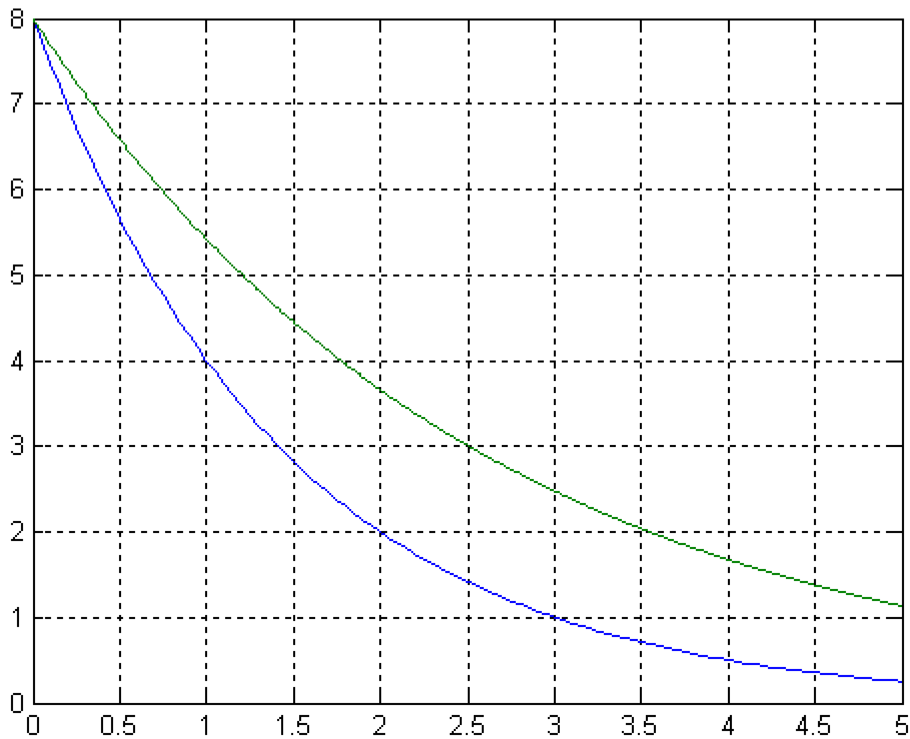
et cela donne :

La courbe bleue de f1(t) est correcte ; mais la courbe verte montre que le coefficient 1E-15 entraîne une dégradation appréciable de la précision de f2(t).

On peut encore construire un tableau "F" contenant les deux fractions précédentes et calculer f1(t) seule, puis f1(t) et f2(t) ensemble. On peut ensuite tracer f1(t) seule, puis extraite du calcul couplé avec f2 (t) :

```
MATLAB :
>> p=tf('p') ;
>> F(1)=8/(log(2)+p) ;
>> F(2)=8/(log(2)+p+1E-15*p^2) ;
>> [x1 t1]=impulse(F(1),5) ;
>> [x  t ]=impulse(F    ,5) ;
>> plot(t1,x1,t,x(:,:,1)) ; grid ;
```

et cela donne :

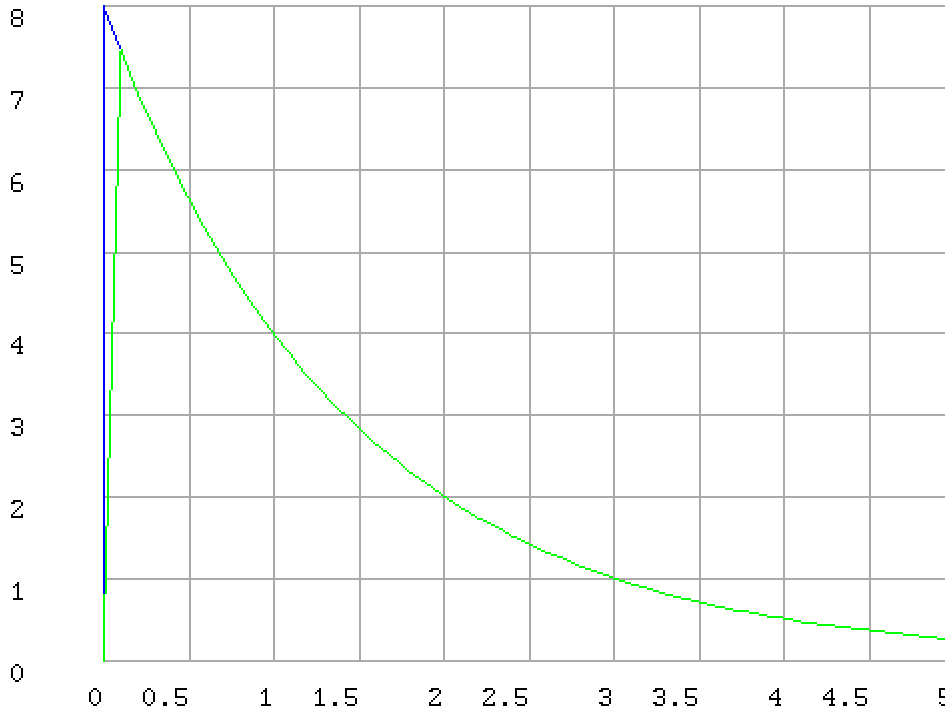


La courbe bleue de f1(t) calculée seule est correcte ; mais la courbe verte montre que la présence de f2(t) dans le calcul dégrade la précision de f1(t). Ici 10^{-15} n'est pas négligeable, cette petite cause produit de grands effets.

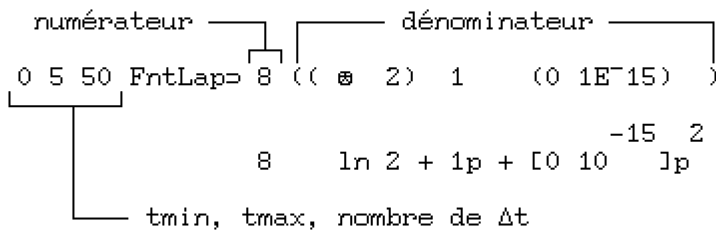
En APL/HUB on fait les deux calculs simultanément :

```
APL/HUB :
       Graf 8 Plot(⊂5 3);0 5 50 FntLap⊃8((⍟2)1(0 1E¯15))
```

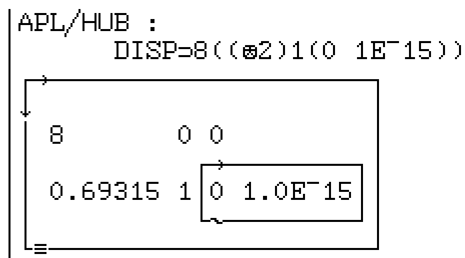
une ligne de code suffit, et cela donne :



Puisque l'intervalle 5 est subdivisé en 50 parties, il y a un point à chaque multiple de 5/50=0.1 ; pour $t \geq 0.1$ la courbe f2(t) se superpose à f1(t) et l'écrase. Le coefficient 1E-15 ne dégrade pas la précision. Le calcul proprement dit s'analyse :



L'argument droit de "FntLap" est un tableau gigogne qui se développe :



c'est une matrice à deux lignes (numérateur, dénominateur) et trois colonnes (second degré).

Au lieu de deux valeurs d'un coefficient on aurait pu en placer dix, cent, mille ; "FntLap" ferait encore les calculs simultanément, tandis qu'en MATLAB il faudrait faire une boucle de programme.

MATLAB ne prend pas assez de précautions pour garantir la précision de certains calculs.

Le temps de calcul (automatique)

Considérons un système d'ordre deux de fonction de transfert :

$$H(p) = \frac{1}{1 + 2\zeta p + p^2}$$

On veut tracer sa réponse à un échelon unité en fonction du coefficient d'amortissement ζ. La transformée de Laplace de cette réponse est :

```
H(p)/p
```

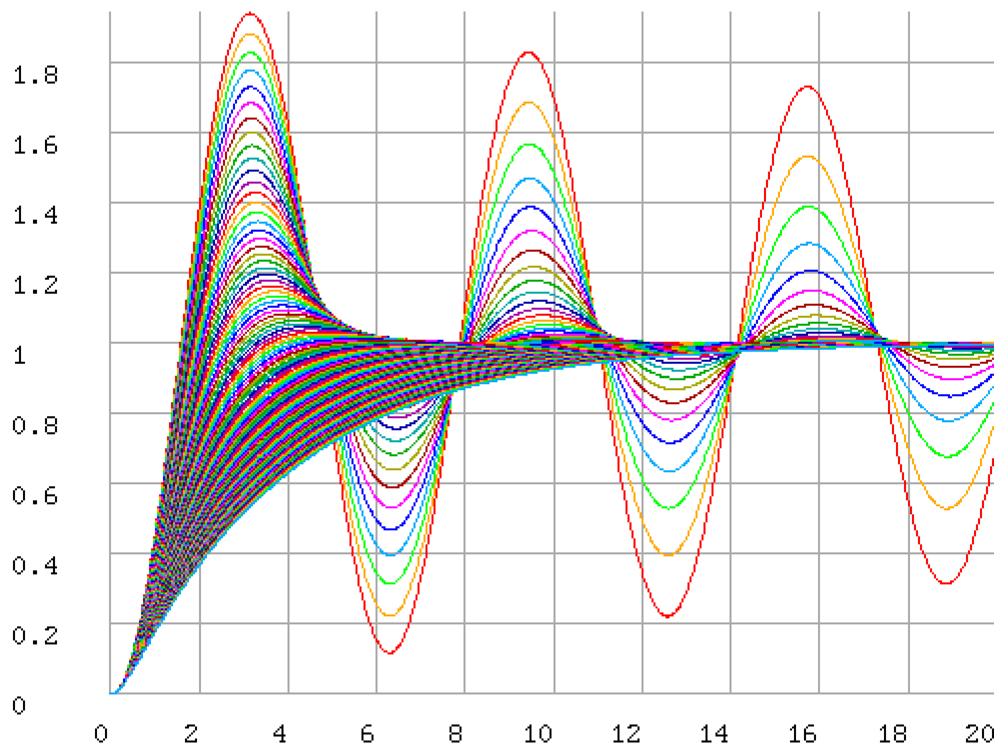
Prenons 100 valeurs de ζ en progression arithmétique :

0.2, 0.4, ... 2

En APL on peut faire :

```
APL/HUB :
        Graf 0 Plot 0 20 1000 FntLap ⊃ 1 (0 1 (2×(1+⍳n)×2÷n←100) 1)
```

et sur un certain ordinateur on obtient après une seconde deux (1.2s) :



En MATLAB, par manque de tableaux gigognes analogues à APL, une manière facile consiste à faire une boucle dans une fonction ; chaque passage dans la boucle fait le calcul pour un ζ et trace la courbe correspondante :

```
MATLAB :
function echord2(z,t)
if nargin==1
    t=[] ;
end
d=[1 1.5 1] ;
hold on ;
for i=1:length(z)
    d(2)=2*z(i) ;
    step(tf(1,d),t) ;
end
grid ;
```

Puis on fait :

```
MATLAB :
>> n=100  ; z=(1:n)*2/n ;
>> m=1000 ; t=(0:m)*20/m ;
>> echord2(z,t)
```

On obtient les mêmes courbes qu'avec APL sur le même ordinateur après soixante-quinze secondes (75s).

Un utilisateur expérimenté de MATLAB a modifié la fonction "echord2" en déplaçant les courbes en dehors de la boucle :

```
MATLAB :
function echord2a(z,t)
if nargin==1
    t=[] ;
end
d=[1 1.5 1] ;
t2=zeros(1,length(z)) ;
y1=zeros(1,length(z)) ;
for i=1:length(z)
    d(2)=2*z(i) ;
    [t1,y,x]=step(tf(1,d),t) ;
    t2(1:length(t1),i)=t1 ;
    y1(1:length(y),i)=y ;
end
plot(y1,t2) ;
grid ;
```

On obtient encore les mêmes courbes sur le même ordinateur après une seconde deux (1.2s) aussi ; un piège.

En APL/HUB, c'est une seule ligne de code sans fonction. En APL/HUB, on peut se mettre dans les mêmes conditions que l'exemple MATLAB qui prend 75s en faisant chaque calcul et chaque courbe dans une boucle, donc dans une fonction ; on obtient le même graphique après une seconde neuf (1.9s) seulement.

Courbe de Black (automatique)

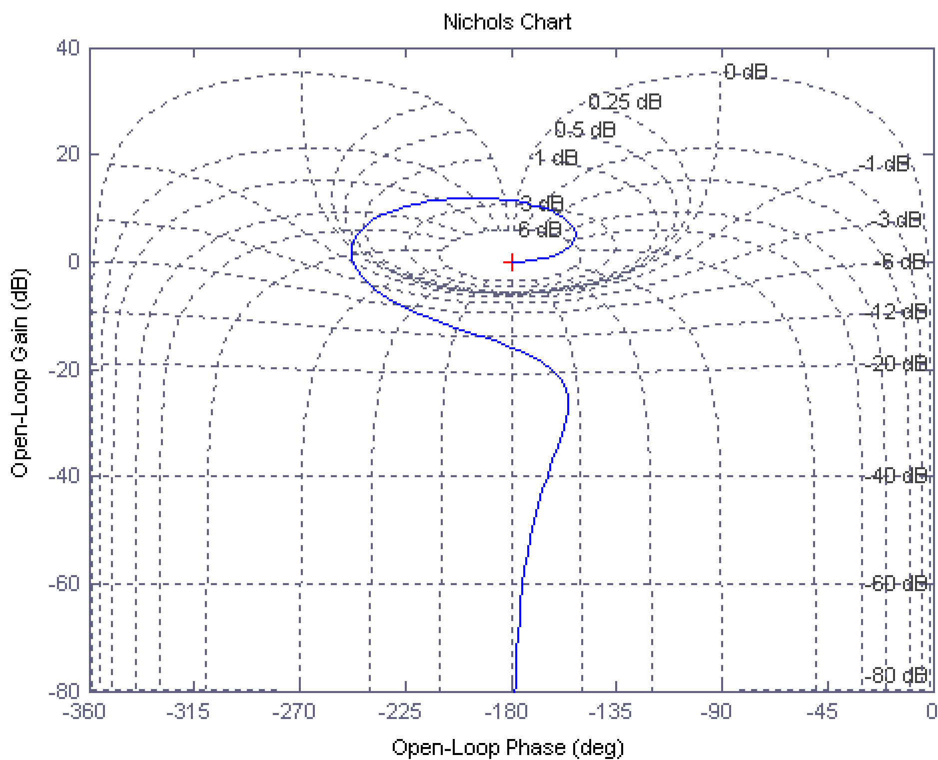
Dans l'étude d'un système asservi, on peut être amené à tracer sa fonction de transfert en boucle ouverte dans le plan de Black, appelé Nichols par MATLAB. Prenons comme exemple la fonction de transfert en boucle ouverte :

$$F(p) = \frac{-1 - 20p - 20p^2}{1 + 5p + 100p^2 - 50p^3 - 10p^4}$$

ce système est instable à cause d'un pôle à partie réelle positive. On trace la courbe de Black par :

```
MATLAB :
>> nichols(tf(-[20 20 1],[-10 -50 100 5 1])) ; grid ;
```

ce qui donne :



Si on multiplie F(p) par un coefficient de gain "K", ce qui translate verticalement la courbe de log K par rapport aux axes, on ne peut pas facilement en déduire le nombre de pôles instables (à parties réelles positives) en boucle fermée en fonction de K. De plus, la courbe est assortie d'une grille en traits interrompus qui permettait autrefois de calculer graphiquement, point par point, la fonction de transfert en boucle fermée. Depuis que l'ordinateur permet de faire ce calcul rapidement, cette grille est obsolète.

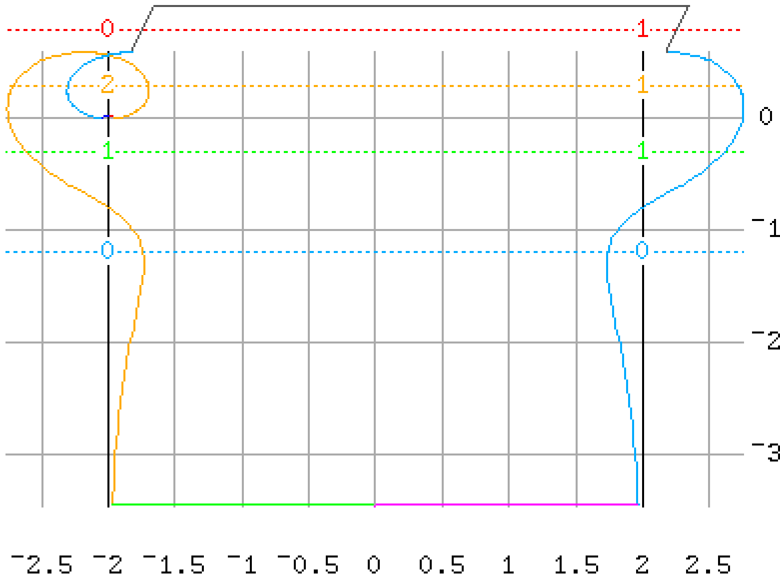
Une amélioration de la courbe de Black consiste à tracer le logarithme (complexe) de la courbe de Nyquist complète par :

```
APL/HUB
Graf ¯0.8 ¯0.3 0.3 1.2 1 BodBlk LieuTrsf⊃(-1 20 20)(1 5 100 ¯50 ¯10)
```

où on a marqué quatre valeurs de K ; l'argument gauche de "BodBlk" s'analyse :

```
¯0.8 ¯0.3 0.3 1.2  1
└───────────────┘ └┬┘
                   │           ┌ 0 = Bode
log décimaux       └── code    │ 1 = Black
des gains                      └ 2 = Bode+Black
```

et cela donne :



Le nombre de pôles instables (à parties réelles positives) en boucle fermée est le nombre de tours de la courbe autour des points critiques pour le réglage du gain ; il est indiqué sur la figure :

```
K = 0.16 => 0+1 = 1  (rouge)
K = 0.5  => 2+1 = 3  (jaune)
K = 2    => 1+1 = 2  (vert)
K = 16   => 0+0 = 0  (turquoise)
```

et on peut vérifier les pôles en boucle fermée arrondis à 10^{-4} près, encore une seule ligne de code pour calculer ces quatre cas :

```
APL/HUB :
      4⍕¨⊂[⍳2]⊃RacPol Col+⌿⊃((⊂0.16 0.5 2 16)×-1 20 20)(1 5 100 ¯50 ¯10)
  ¯0.0114  0.0919    0.0270 ¯0.0707    ¯0.0273  0.0000    ¯0.0493  0.0000
  ¯0.0114 ¯0.0919    0.0270  0.0707     0.5541 ¯0.5441    ¯1.9076  0.0000
   1.5107  0.0000    1.3613  0.0000     0.5541  0.5441    ¯1.5216 ¯3.6927
  ¯6.4880  0.0000   ¯6.4152  0.0000    ¯6.0809  0.0000    ¯1.5216  3.6927
⍝     Re       Im        Re       Im         Re       Im         Re       Im

⍝  1  Re>0             3  Re>0             2  Re>0             0  Re>0
```

L'argument droit de "LieuTrsf" se développe :

```
APL :
      ⊃(-1 20 20)(1 5 100 ¯50 ¯10)
 ¯1 ¯20 ¯20   0   0
  1   5 100 ¯50 ¯10
```

c'est une matrice à deux lignes : numérateur, dénominateur.

Le "nichols" de MATLAB ne trace que la partie jaune de "BodBlk" d'APL/HUB ; c'est la programmation moderne d'une technique obsolète.

Historique

J'ai abandonné le fortran en 1972 pour essayer puis adopter l'APL. Il y avait une différence

colossale entre les deux langages. Beaucoup de gens étaient adversaires de l'APL, la majorité des informaticiens. Depuis cette époque, beaucoup d'informaticiens semblent manifester une farouche hostilité à l'APL. Pourquoi ? Diverses explications sont possibles, mais ce ne sont que des hypothèses.

La programmation en APL bouleversait les méthodes de l'informatique de l'époque. Par exemple, il ne fallait plus parcourir les cases d'un tableau pour les traiter une par une dans des boucles emboîtées, car APL fournissait des primitives efficaces sur tableaux. Les informaticiens n'y retrouvaient pas ce qu'ils avaient appris à l'école et, la programmation étant simplifiée, ils pouvaient craindre que des clients potentiels n'aient plus besoin de leurs services.

Les fonctions APL n'étaient pas compilées, mais interprétées ; et une règle disait que les programmes compilés s'exécutent plus vite que ceux interprétés. Ce n'était pas vrai pour toutes les fonctions APL, et on oubliait le temps passé par l'homme à l'analyse et à la programmation. Comparés à ceux d'aujourd'hui, les ordinateurs de l'époque (grosses armoires en salles climatisées) étaient lents et coûteux. Depuis, les méthodes d'interprétation se sont améliorées. Le travail en APL était globalement efficace, et ceux qui l'utilisaient couramment refusaient de revenir au fortran. Comme conséquence de cette hostilité, quand les décideurs d'une entreprise doivent acheter un logiciel, ils demandent conseil à leurs informaticiens qui ne leur recommandent surtout pas APL.

Vers 1990 on commence à entendre parler de MATLAB. Je me demande bien pourquoi ceux qui l'ont créé ont voulu refaire ce qui existait déjà en plus puissant, si ce n'est l'hostilité déjà signalée ; car APL connaissait les tableaux gigognes et un ensemble de concepts et de primitives permettant de les traiter efficacement par des programmes concis.

Bien entendu il existe des algorithmes d'analyse numérique en fortran, C ou MATLAB. Il en existe aussi en APL, nous en avons vu dans ce texte. Pour ceux qui n'existent pas en APL, il faut savoir que les diverses réalisations d'APL fournissent des passerelles entre APL et des algorithmes en binaire exécutable, qu'ils proviennent de fortran, C ou assembleur. De plus il faut comprendre qu'on peut traduire en APL des programmes fortran, C ou MATLAB sans trop de difficulté, car la version APL est toujours beaucoup plus simple. Naturellement cela représente un peu de travail ; mais il en vaut le coût. Ce faisant, on peut étendre le domaine d'application des algorithmes APL ainsi construits au-delà de ce qu'on peut espérer en fortran, C ou MATLAB. C'est la transformation inverse qui est compliquée, voire impossible.

Conclusion

MATLAB semble dériver du fortran, dont il a supprimé les déclarations de type et de dimension, auquel il a incorporé des matrices obligées.

Il essaie de forcer les données à être matricielles : par exemple il place les coefficients d'un polynôme dans une matrice à une ligne, alors que la notion de matrice n'a rien à voir avec celle de polynôme. Il fournit un grand nombre d'algorithmes de traitement réservé aux matrices. Mais si un problème ne peut pas être mis sous cette forme (arborescences, valeurs multiples d'une donnée, etc...) il faut faire une programmation du genre fortran.

Il y a des pièges dans fortran, il y en a aussi dans MATLAB mais ce ne sont toujours pas les mêmes.

Il y a plusieurs structures de données en MATLAB ; en APL une seule structure suffit pour en faire davantage.

Il y a plusieurs syntaxes en MATLAB suivant les fonctions ; en APL une seule syntaxe suffit, simplifiant la programmation.

Le choix des symboles de fonctions est plus restreint en MATLAB qu'en APL.

MATLAB place chaque fonction dans un fichier ; APL place toutes les fonctions en mémoire vive.

Les variables et fonctions d'APL s'organisent dans une structure arborescente dynamique ; dans MATLAB il n'y a que deux niveaux : privé et public.

Les fonctions qui font la même chose sont plus longues en MATLAB qu'en APL, quelquefois de beaucoup.

Une fonction MATLAB programmée en imitant APL peut en remplacer plusieurs tout en étant plus tolérante.

APL sait faire sans programmation des opérations que MATLAB ne peut faire qu'avec de longues fonctions.

Certains calculs numériques de MATLAB peuvent être faux, le choix de la méthode n'étant probablement pas le bon.

Certains calculs peuvent prendre plus de temps en MATLAB qu'en APL.

MATLAB peut faire des calculs qui n'ont aucun sens.

Certaines primitives de MATLAB sont superflues en APL.

Certaines primitives d'APL n'ont pas d'équivalent dans MATLAB.

......

Arrêtons là l'énumération ; bref, APL est plus simple, plus concis, plus puissant, plus cohérent et plus précis que MATLAB.

Pour ces raisons, et bien d'autres, je continue à utiliser APL et non pas MATLAB, qui compliquerait mon travail.

Le site :

http://www.afapl.asso.fr

contient des fonctions APL, des articles sur APL, ainsi qu'une présentation d'APL.